Model-Based Testing with Graph Grammars

MSc Thesis *(Afstudeerscriptie)*

written by

Vincent de Bruijn

Formal Methods & Tools,
University of Twente,
Enschede,
The Netherlands

v.debruijn@student.utwente.nl

January 1, 2013

Abstract

Graph Grammars describe system behavior through graphs and graph transformation rules. Graph Grammars have not been used for Model-Based Testing. However, Graph Grammars have many structural advantages, which are potential benefits for the model-based testing process. We describe a model-based testing setup with Graph Grammars. The result is a system for automatic test generation from Graph Grammars. A graph transformation tool, GROOVE, and a model-based testing tool, ATM, are used as the backbone of the system. The system is validated using the results of several case studies.

The result of this report is a tool, GRATiS, which can be used for model-based testing on Graph Grammars.

hier komt ook nog een kort overzicht van de conclusies

Contents

Chapter 1

Introduction

In this introduction, first the importance of testing and automation of testing is stressed. Then Model-Based Testing is shown to be a useful tool for automation of testing. Graph Grammars and graph transformation are argued to be useful as formalism for Model-Based Testing. Some leading tools for automatic test generation are set out, which include the tools used in this report. The research goals are given and finally a roadmap explains the basic structure of the rest of this report.

1.1 Testing

In software development projects, often time and budget costs are exceeded. Laird and Brennan [10] investigated in 2006 that 23% of all software projects are canceled before completion. Furthermore, of the completed projects, only 28% are delivered on time with the average project overrunning the budget with 45%. The cause of this often are the unclear ambigious requirements of the software system to develop.

Testing is an important part of software development, because it decreases future maintainance costs [16]. Testing is a complex process and should be done often [19]. Therefore, the testing process should be as efficient as possible in order to save resources.

Test automation allows repeated testing during the development process. The advantage of this is that bugs are found early and can therefore be fixed early. A widely used practice is maintaining a *test suite*, which is a collection of test-cases. However, when the creation of a test suite is done manually, this still leaves room for human error [13]. The process of deriving tests tends to be unstructured, barely motivated in the details, not reproducible, not documented, and bound to the ingenuity of single engineers [27].

1.2 Model-based Testing

The existence of an artifact that explicitly encodes the intended behaviour can help mitigate the implications of these problems. Creating an abstract representation or a *model* of the system is an example of such an artifact. What is meant by a model in this report, is the description of the behavior of a system. In particular, the term model will be often used to describe transition-based notations, such as finite state machines, labelled transition systems and I/O automata. Other notations, such as UML statecharts, are not considered as models in this report.

A model can be used to systematically generate tests for the system. This is referred to as *model-based testing*. Generating tests automatically leads to a larger test suite than if done manually. A large, systematically built test suite is bound to find more bugs than a smaller, manually built one.

Models are created from the specification documents provided by the end-user. These specification documents are 'notoriously error-prone' [15]. This implies that the model itself needs validation. Validating the model usually means that the requirements themselves are scrutinised for consistency and completeness [27]. This helps to clear up ambigious requirements early on, which allows better estimation of the budget and time demands.

The stakeholders evaluate the constructed model to verify its correctness. However, the visual or textual representation of large models may become troublesome to understand, which is referred to as the model having a low model transparency or high model complexity. The problem with transition systems is that a larger number of states and/or transitions decreases the model transparency. We think that low model transparency make errors harder to detect and that it obstructs the feedback process of the stakeholders. Using models with high transparency is therefore essential.

1.3 Graph Transformation

A formalism that claims to have more model transparency is Graph Transformation. The system states are represented by graphs and the transitions between the states are accomplished by applying graph change rules to those graphs. These rules can be expressed as graphs themselves. A graph transformation model of a software system is therefore a collection of graphs, each a visual representation of one aspect of the system. This formalism may therefore provide a more intuitive approach to system modelling than traditional state machines. Graph Transformation and its potential benefits have been studied since the early '70s. The usage of this computational paradigm is best described by the following quote from Andries et al. [1]:

> Graphs are well-known, well-understood, and frequently used means to represent system states, complex objects, diagrams, and networks, like flowcharts, entity-relationship diagrams, Petri nets, and many more. Rules have proved to be extremely useful for describing computations by local transformations: Arithmetic, syntactic, and deduction rules are well-known examples.

An informative paper on graph transformations is written by Heckel et al. [8]. A quote from this paper:

> Graphs and diagrams provide a simple and powerful approach to a variety of problems that are typical to computer science in general, and software engineering in particular.

1.4 Tools

Tools for automatic test generation already exist. The testing tool developed by Axini[1] is used for the automatic test generation on *symbolic* models, which combine a state and data type oriented approach. This tool is used in this report and is referred to as Axini Test Manager (ATM). In Utting et al. [27], a taxonomy is done on different model-based testing tools:

- TorX [24]: accepts behaviour models such as I/O labelled transition systems. A version of this tool written in Java under continuous development is JTorX [2]. This version accepts the same kind of models as ATM.

[1] http://www.axini.nl/

- Spec Explorer[28]: provides a model editing, composition, exploration and visualization environment within Visual Studio, and can generate offline .NET test suites or execute tests as they are generated (online).
- JUMBL[20]: an academic model-based statistical testing that supports the development of statistical usage-based models using Markov chains, the analysis of models, and the generation of test cases.
- AETG[5]: implements combinatorial testing, where the number of possible combinations of input variables are reduced to a few 'representative' ones.
- STG tool[4]: implements conformance testing techniques to automatically derive symbolic test cases from formal operational specifications.

The graph transformation tool GROOVE[2] is used to model and explore graph grammars.

Zijn er nog andere graph transformation tools?

1.5 Research goals

The motivation above is given for using graph grammars as a modelling technique. The goal of this research is to create a system for automatic test generation on graph grammars. If the assumptions that graph grammars provide a more intuitive modelling and testing process hold, this new testing approach will lead to a more efficient testing process and fewer incorrect models. The system to be designed, once implemented and validated, should provide a valuable contribution to the testing paradigm. The tools GROOVE and ATM are used to create this system.

The research goals are split into a design and validation component:

1. **Design**: Design and implement a system using ATM and GROOVE which performs model-based testing on graph grammars.
2. **Validation**: Validate the design and implementation using case studies and performance measurements.

The result of the design goal is one system called the GROOVE-Axini Testing System (GRATiS). The validation goal uses case-studies with existing specifications from systems tested by Axini. Each case-study has a graph grammar and a symbolic model which describe the same system. GRATiS and ATM are used for the automatic test generation on these models respectively. Both the models and the test processes are compared as part of the validation.

The solution has to uphold three requirements:

1. A graph grammar must be used as the model; it must derive from the specification and be used for the testing.
2. It must be possible to measure the test progress/completion, by means of *coverage* statistics (explained in detail in section 2.1.2).
3. The solution must be efficient: it should be usable in practice, therefore the technique should be scalable and the imposed constraints reasonable from a practical view point.

1.6 Roadmap

This report has five more chapters: first, the concepts described in this chapter are elaborated in chapter 2. The design of GRATiS is described in chapter ??. The implementation of GRATiS is

[2] http://sourceforge.net/projects/groove/

covered in chapter **??**. The validation of GRATiS is in chapter **??**. Finally, conclusions are drawn in chapter **??**.

hoofdstukken met ?? zitten niet in dit verslag

Chapter 2

Background

The structure of this chapter is as follows: the general model-based testing process is set out in section 2.1. Some basic concepts from algebra are described in section 2.2. Then the symbolic models used by ATM are described in section 2.3. Section 2.4 describes the Graph Grammar formalism. GROOVE and ATM are described in section 2.5.

2.1 Model-based Testing

Model-based testing is a testing technique where a System Under Test (SUT) is tested for conformance to a model description of the system. The general setup for this process is depicted as a UML sequence diagram in Figure 2.1. The specification of a system is provided as a model to a test derivation component which generates a test suite. The test suite is used by a test execution component to test the SUT. Tests are executed by providing input/stimuli to the SUT and monitoring the output/response. The test execution component evaluates whether the correct responses are given. It gives a 'pass' or 'fail' verdict depending on whether the SUT conforms to the model or not.

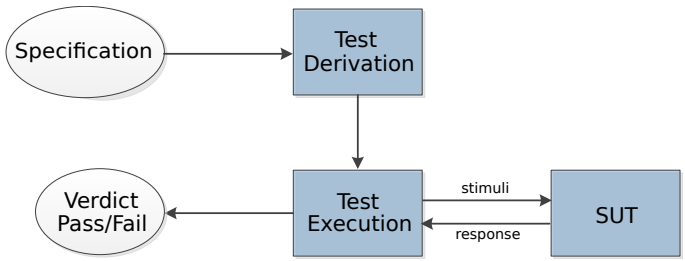


Figure 2.1: A general model-based testing setup

This type of model-based testing is called *batch testing* or *offline testing*. Another type of model-based testing is *on-the-fly* testing. The main difference is that no test cases are derived, instead a transition in the model is chosen and tested on the system directly. The general architecture for this process is shown in Figure 2.2. An example of an on-the-fly testing tool is TorX [24].

Variations of state machines and transition systems have been widely used as the underlying model for test generation. Other tools use the structure of data types to generate test data.

The stucture of the rest of this section is as follows. First, previous work on model-based testing is given. Then, Labelled Transition Systems are introduced. This is a basic formalism useful to

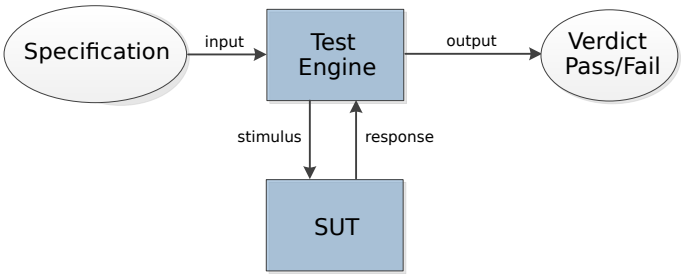


Figure 2.2: A general 'on-the-fly' model-based testing setup

understand the models in the rest of the paper, namely Graph Grammars and Symbolic Transition Systems. Next, an adaptation of Labelled Transition Systems, the Input-Output Transition System is described. This is a useful formalism for Model-Based Testing. Finally, the notion of *coverage* is explained.

2.1.1 Previous work

Formal testing theory was introduced by De Nicola et al. [18]. The input-output behavior of processes is investigated by series of tests. Two processes are considered equivalent if they pass exactly the same set of tests. This testing theory was first used in algorithms for automatic test generation by Brinksma [3]. This led to the so-called *canonical tester* theory. Tretmans gives a formal approach to protocol conformance testing (whether a protocol conforms to its specifications) in [25] and an algorithm for deriving a sound and exhaustive test suite from a specification in [26]. A good overview of model-based testing theory and past research is given in "Model-Based Testing of Reactive Systems" [14].

2.1.2 Labelled Transition Systems

A Labelled Transition System (LTS) is a structure consisting of states with labelled transitions between them.

Definition 2.1.1. Labelled Transition Systems
An LTS is a 4-tuple $\langle Q, L, T, q_0 \rangle$, where:

- Q is a finite, non-empty set of states
- L is a finite set of labels
- $T \subseteq Q \times (L \cup \{\tau\}) \times Q$, with $\tau \notin L$, is the transition relation
- $q_0 \in Q$ is the initial state.

We write $q \xrightarrow{\mu} q'$ if there is a transition labelled μ from state q to state q', i.e., $(q, \mu, q') \in T$. q, q' are called the source and target states of the transition respectively. The informal idea of such a transition is that when the transition system is in state q it may perform action μ, and go to state q'.

Input-Output Transition Systems A useful type of transition system for model-based testing is the Input-Output Transition System (IOTS) by Tretmans [26]. Assuming that implementations communicate with their environment via inputs and outputs, this formalism is useful for describing system behavior. The system is regarded as a black box and the IOTS specifies the allowed inputs and outputs.

IOTSs have the same definition as LTSs with one addition: each label $l \in L$ has a type $\iota \in Y$, where $Y = \{input, output\}$. Each label can therefore specify whether the action represented by the label is a possible input or an expected output of the system under test. An IOTS is formally defined as:

Definition 2.1.2. Input-Output Transition Systems

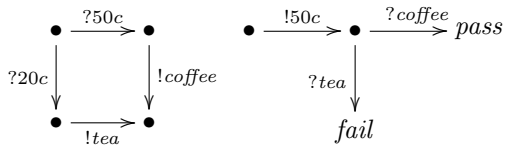
- Q
- L
- $T_Y \subseteq T \times Y$
- q_0

When the transition system is in the source state of an input transition, the input can be given to the SUT. When the transition system is in the source state of an output transition, the output should be observed from the SUT. In both cases, the transition system advances to the target state of the transition. The case where a state has both input and output transitions is not considered in this report.

An example of such an IOTS is shown in Figure 2.3a. This system allows an input of 20 or 50 cents and then outputs tea or coffee accordingly. The inputs are preceded by a '?', the outputs are preceded by an '!'. This system is a specification of a coffee machine. A test case can also be described by an IOTS with special pass and fail states.

A test case for the coffee machine is given in Figure 2.3b. The test case shows that when an input of '50c' is given, an output of 'coffee' is expected from the tested system, as this results in a 'pass' verdict. When the system responds with 'tea', the test case results in a 'fail' verdict. The pass and fail verdicts are two special states in the test case, which are sink states, i.e., once in either of those the test case cannot leave that state.

Test cases should always reach a pass or fail state within finite time. This requirement ensures that the testing process halts.



(a) An IOTS (b) An IOTS test case

Figure 2.3: The specification of a coffee machine and a test case as an IOTS

Coverage The number of tests that can be generated from a model is potentially infinite. Therefore, there must be a test selection strategy to maximize the quality of the tests while minimizing the time spent testing. Coverage statistics help with test selection. Such statistics indicate how much of the model is tested. A transition in the model is considered tested when the transition system is in the source state of the transition and either the input is given to or the output is observed from the SUT, depending on the type of the transition.

When the SUT is a black box, typical coverage metrics are state and transition coverage of the model [12, 17, 7]. State coverage can be measured by dividing the tested states by the total states in the model. The same process applies to transition coverage. As an example, the coverage metrics of the IOTS test case example in 2.3b are calculated. The test case tests one path through the specification and passes through 3 out of 4 states and 2 out of 4 transitions. The state coverage is therefore 75% and the transition coverage is 50%.

Coverage statistics are calculated to indicate how adequately the testing has been performed [29]. These statisics are therefore useful metrics for communicating how much of a system is tested.

2.2 Algebra

Some basic concepts from algebra are described here. For a general introduction into logic we refer to [9]. This section explains in order: multi-sorted signatures, algebrae, variables & terms and term-mapping & valuations. The algebra described here will be used in the next sections to formally define Symbolic Transition Systems and Graph Grammars.

Definition 2.2.1. Multi-sorted Signatures
A *multi-sorted signature* $\langle S, F\rangle$ describes the sorts and function symbols of a formal language. S is a set of sorts. F is a set of function symbols. Each $f \in F$ has an arity $n \in \mathbb{N}$, where a function symbol with arity $n = 0$ is called a constant symbol. F^i denotes the subset of F with function symbols of arity $n = i$. The sort of a function symbol $f \in F$ with arity n is given by $\sigma(f) = s_1...s_{n+1}$, with $s_i \in S$ for $1 \leq i \leq n$. s_{n+1} is the return sort. In this report, $S = \{int, real, bool, string\}$ denoting the integer, real, boolean and string sorts respectively. F features the commonly used function symbols, which include, but not restricted by, '$+$', '$*$', '$=$', '$<$', '0', '1'.

Definition 2.2.2. Algebrae
An *algebra* $\mathcal{A} = \langle \mathbb{U}, \Phi\rangle$ has a non-empty set $\mathbb{U}$ of values called a *universe*, partitioned into $\mathbb{U}^s$ for each $s \in S$, and a set Φ of functions. A function $\phi_{\mathcal{A}}$ is typed $\mathbb{U}^{s_1}_{\mathcal{A}} \times ... \mathbb{U}^{s_n}_{\mathcal{A}} \to \mathbb{U}^{s_{n+1}}_{\mathcal{A}}$, where $s_1...s_{n+1}$ is the sort of the function symbol given by the signature. For example, $<_{\mathcal{A}}$: $\mathbb{U}^{int}_{\mathcal{A}} \times \mathbb{U}^{int}_{\mathcal{A}} \to \mathbb{U}^{bool}_{\mathcal{A}}$ represents the 'less-than' comparison of two integers.

Definition 2.2.3. Variables
We define $\mathcal{V} = \mathcal{V}^{int} \uplus \mathcal{V}^{real} \uplus \mathcal{V}^{bool} \uplus \mathcal{V}^{string}$ to be the set of *variables*. *Terms* over V, denoted $\mathcal{T}(V)$, are built from function symbols F and variables $V \subseteq \mathcal{V}$. The definition of a term is:

$$\begin{array}{rcll} t & ::= & f(t_1...t_n) & , \textit{where n is the arity of } \phi \\ & | & x & , \textit{where } x \textit{ is a variable.} \end{array}$$

We write $var(t)$ to denote the set of variables appearing in a term $t \in \mathcal{T}(V)$. Terms $t \in \mathcal{T}(\emptyset)$ are called ground terms. An example of a term t is $(x + (y - 1))$, with $var(t) = \{x, y\}$. The type of a term is given by:

$$\begin{array}{rll} \sigma : t \mapsto & s & \textit{if } t = x \in \mathcal{V}^s \\ & s_{n+1} & \textit{if } t = f(t_1...t_n) \textit{ and } \sigma(f) = s_1...s_{n+1}, \textit{ provided } \sigma(t_i) = s_i \end{array}$$

The set of terms with return type *bool*, is denoted as $\mathcal{B}(\mathcal{V})$. An example is $(x < y)$, where the result is *true* or *false*.

Definition 2.2.4. Term-mapping
A *term-mapping* is a function $\mu : \mathcal{V} \to \mathcal{T}(\mathcal{V})$. A *valuation* ν is a function $\nu : \mathcal{V} \to \mathbb{U}$ that assigns values to variables. For example, given an algebra, $\nu : \{(x \mapsto 1), (y \mapsto 2)\}$ assigns the values 1 and 2 to the variables x and y respectively. A valuation of a term given $\mathcal{A}$ is defined by:

$$\begin{array}{rlcl} \nu : & x & \mapsto & \nu(x) \\ & f(t_1...t_n) & \mapsto & f_{\mathcal{A}}(\nu(t_1)...\nu(t_n)) \end{array}$$

When every variable in a term is defined by a valuation, the term can be valuated to a value. Therefore, when every variable in a term-mapping is defined by a valuation, a new valuation can be obtained. Formally, this is defined as: $\_after\_ : (\mathcal{V} \to \mathbb{U}) \times (\mathcal{V} \to \mathcal{T}(\mathcal{V})) \to (\mathcal{V} \to \mathbb{U})$. Given a valuation ν and a term-mapping μ, $(\nu \textit{ after } \mu) : \nu \mapsto \nu(\mu(\nu))$.

2.3 Symbolic Transition Systems

Symbolic Transition Systems (STSs) combine a state oriented and data type oriented approach. This formalism is a specification of system behavior like LTSs. These systems are used in practice in ATM and will therefore be part of GRATiS. In this section, previous work on STSs is reviewed. The definitions of STSs and IOSTSs follow. An example of an IOSTS is then given. Next, the mapping of an STS to an LTS is explained and illustrated by an example. This mapping is useful when comparing STSs to Graph Grammars, because both systems can be mapped to an LTS and then compared. Finally, different coverage metrics on STSs are explained.

2.3.1 Previous work

STSs are introduced by Frantzen et al. [11]. This paper includes a detailed definition, on which the definition below is based. The authors also give a sound and complete test derivation algorithm from specifications expressed as STSs. Deriving tests from a symbolic specification or *Symbolic test generation* is introduced by Rusu et al. [23]. Here, the authors use *Input-Output Symbolic Transition Systems* (IOSTSs). These systems are very similar to the STSs in [11]. However, the definition of IOSTSs we will use in this report is based on the STSs by [11]. A tool that generates tests based on symbolic specifications is the STG tool, described in Clarke et al. [4].

2.3.2 Definition

An STS has *locations* and *switch relations*. If the STS represents a model of a software system, a location in the STS represents a state of the system, not including data values. A switch relation defines the transition from one location to another. The *location variables* are a representation of the data values in the system. A switch relation has a *gate*, which is a label representating the execution steps of the system. Gates have *interaction variables*, which represent some input or output data value. Switch relations also have *guards* and *update mappings*. A guard is a term $t \in \mathcal{B}(\mathcal{V})$. The guard disallows using the switch relation when the valuation of the term results in *false*. When the valuation results in *true*, the switch relation of the guard is *enabled*. An update mapping is a term-mapping of location variables. After the system switches to a new location, the variables in the update mapping will have the value corresponding to the valuation of the term.

Definition 2.3.1. Symbolic Transition Systems
A Symbolic Transition System is a tuple $\langle W, w_0, \mathcal{L}, \imath, \mathcal{I}, \Lambda, D \rangle$, where:

- W is a finite set of locations and $w_0 \in W$ is the initial location.
- $\mathcal{L} \subseteq \mathcal{V}$ is a finite set of location variables.
- $\imath$ is a term-mapping $\mathcal{L} \to \mathcal{T}(\emptyset)$, representing the initialisation of the location variables.
- $\mathcal{I} \subseteq \mathcal{V}$ is a set of interaction variables, disjoint from $\mathcal{L}$.
- Λ is a finite set of gates. The unobservable gate is denoted $\tau (\tau \notin \Lambda)$; we write Λ_τ for $\Lambda \cup \{\tau\}$. The arity of a gate $\lambda \in \Lambda_\tau$, denoted $arity(\lambda)$, is a natural number. The parameters of a gate $\lambda \in \Lambda_\tau$, denoted $param(\lambda)$, are a tuple of length $arity(\lambda)$ of distinct interaction variables. We fix arity$(\tau) = 0$, i.e. the unobservable gate has no interaction variables.
- $D \subseteq W \times \Lambda_\tau \times \mathcal{B}(\mathcal{L} \cup \mathcal{I}) \times (\mathcal{L} \to \mathcal{T}(\mathcal{L} \cup \mathcal{I})) \times W$, is the switch relation. We write $w \xrightarrow{\lambda, \gamma, \rho} w'$ instead of $(w, \lambda, \gamma, \rho, w') \in D$, where γ is referred to as the guard and ρ as the update mapping. We require $(var(\gamma) \cup var(\rho)) \subseteq (\mathcal{L} \cup param(\lambda))$. We define $out(w) \subseteq D$ to be the outgoing switch relations from location w.

An IOSTS can now easily be defined. The same difference between the labels in LTSs and IOTSs apply, namely each gate has a type $\iota \in Y$. As with labels, each gate is preceded by a '?' or '!' to indicate whether it is an input or an output respectively. The full definition as as follows:

Definition 2.3.2. Input-Output Symbolic Transition Systems

- W
- $\mathcal{L}$
- $\imath$
- $\Lambda_Y \subseteq \Lambda \times Y$
- D

2.3.3 Example

In Figure 2.4 the IOSTS of a simple board game is shown, where two players consecutively throw a die and move along four squares. The 'init' switch relation is a graphical representation of the variable initialization $\imath$. The values in the tuple of the IOSTS are defined as follows:

$$
\begin{array}{lcl}
W & = & \{t, m\} \\
w_0 & = & t \\
\mathcal{L} & = & \{T, P1, P2, D\} \\
\imath & = & \{T \mapsto 0, P1 \mapsto 0, P2 \mapsto 2, D \mapsto 0\} \\
\mathcal{I} & = & \{d, p, l\} \\
\Lambda & = & \{?throw, !move\} \\
D & = & \{t \xrightarrow{?throw, 1<=d<=6, D \mapsto d} m, \\
 & & \quad m \xrightarrow{!move, T=1 \wedge l=(P1+D)\%4, P1 \mapsto l, T \mapsto 2} t, \\
 & & \quad m \xrightarrow{!move, T=2 \wedge l=(P2+D)\%4, P2 \mapsto l, T \mapsto 1} t\}
\end{array}
$$

The variables $T, P1, P2$ and D are the location variables symbolizing the player's turn, the positions of the players and the number of the die thrown respectively. The output gate $!move$ has $param = \langle p, l \rangle$ symbolizing which player moves to which location. The input gate $?throws$ has $param = \langle d \rangle$ symbolizing which number is thrown by the die. The switch relation with gate $?throws$ has the restriction that the number of the die thrown is between one and six and the update sets the location variable D to the value of interaction variable d. The switch relations with gate $!move$ have the restriction that it must be the turn of the player moving and that the new location of the player is the number of steps ahead as thrown by the die. The update mapping sets the location of the player to the correct value and passes the turn to the next player. In Figure 2.4 the gates, guards and updates are separated by pipe symbols '|' respectively.

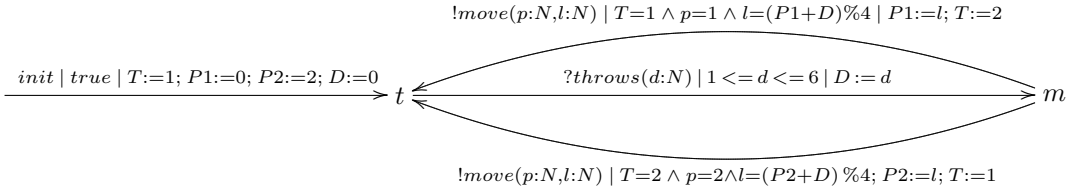


Figure 2.4: The STS of a board game example

2.3.4 STS to LTS mapping

This section shows the method of mapping an STS to an LTS.

Consider an STS J. We can construct an LTS K from J, such that K is an expansion of J. There exists a mapping from the location and location variable valuations to the states of K and from the switch relations and variable valuations of J to the transitions of K. These relations are defined as follows:

Definition 2.3.3. STS to LTS mapping

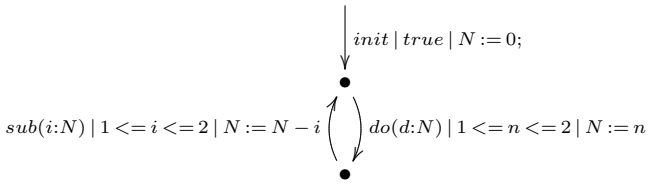
$\mu_Q : \quad (W \times (\mathcal{L} \to \mathbb{U})) \to Q$

$\mu_L : \quad (\Lambda \times (\mathcal{I} \to \mathbb{U})) \to L$

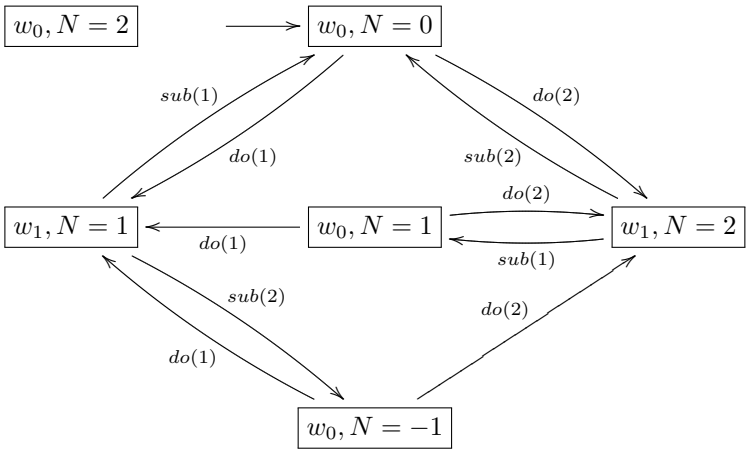
$\mu_T : \quad (w \xrightarrow{\lambda,\gamma,\rho} w', \nu : ((\mathcal{L} \cup \mathcal{I}) \to \mathbb{U})) \mapsto (\mu_Q(w, \nu \restriction \mathcal{L}) \xrightarrow{\mu_L(\lambda,\nu\restriction\mathcal{I})} \mu_Q(w', \nu \; after \; \rho))$

When the number of possible valuations for $\mathcal{L}$ and $\mathcal{I}$ is finite, the transformation is always possible to an LTS with finite number of states.

An example of this transformation is shown in Figure 2.5. The label 'do(1)' in the LTS is a textual representation of the gate 'do' plus a valuation of the interaction variable 'd'. The text on the nodes indicate from which location and valuation the state was created. The node labelled '$w_0, N = 2$' is an example of an unreachable state.



(a) The STS



(b) The LTS

Figure 2.5: An example of a transformation of an STS to an LTS

2.3.5 Coverage

Coverage metrics do not only apply to LTSs, they can also be used on STSs. The simplest metric to describe the coverage of an STS is the location and switch-relation coverage, which express the percentage of locations and switch relations tested. Measuring state and transition coverage of an STS is possible using the LTS from the STS to LTS mapping.

2.4 Graph Grammars

A *Graph Grammar* (GG) is also a specification of system behavior like LTSs and STSs are. A GG is composed of a set of graph transformation rules. These rules indicate how a graph can be transformed to a new graph. These graphs are called *host graphs*. The rules are composed of graphs themselves, which are called *rule graphs*.

The rest of this section is ordered as follows: first, graphs, host graphs, rule graphs and graph transformation rules are explained. Then, the definition of a *Graph Transition System* (GTS) is given. An example of a GG and a GTS is then given. Finally, the definition of IOGGs is given. For a more detailed overview of GGs, we refer to [21, 8, 1].

Definition 2.4.1. Graphs
A *graph* is composed of nodes and edges. In this report, we assume a universe of nodes $\mathbb{V} = \mathbb{W} \uplus \mathbb{U} \uplus \mathcal{V} \uplus 2^{\mathcal{T}}$, where $\mathbb{W}$ is the universe of standard graph nodes. $\mathbb{E}$ is the universe of edges between two nodes in $\mathbb{V}$.

Definition 2.4.2. Host graphs
A host graph G is a tuple $\langle V_G, E_G \rangle$, where:

- $V_G \subseteq (\mathbb{W} \uplus \mathbb{U})$ is the node set of G
- $E_G \subseteq (V_G \backslash \mathbb{U} \times L \times V_G)$ is the edge set of G

Figure 2.6 shows an example of a host graph. Here, $n_1, n_2 \in \mathbb{W}$ are the *identities* of the nodes. The other four nodes are values in $\mathbb{U}$.

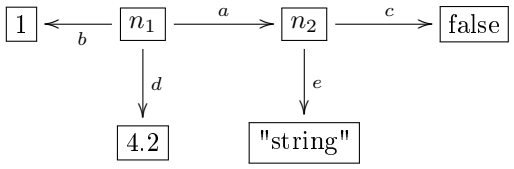


Figure 2.6: An example of a host graph

Definition 2.4.3. Rule graphs
A rule graph H is a tuple $\langle V_H, E_H \rangle$, where:

- $V_H \subseteq (\mathbb{V} \backslash \mathbb{U})$ is the node set of H
- $E_H \subseteq (V_H \times L \times V_H)$ is the edge set of H

In addition, the following must hold:

- $\forall z \in V_H \cap z \in 2^{\mathcal{T}} \,.\, var(z) \subseteq V_H$ - The variables used in the terms must be present as nodes in the rule graph.
- $\forall z \in V_H \cap z \in \mathcal{V} \,.\, \exists(\_, \_, z) \in E_H$ - If a variable is used in a rule graph, it needs context. Therefore, there must be an edge with the variable node as target.

Figure 2.7 shows an example of a rule graph. Here, $r_1, r_2 \in \mathbb{W}$ are the node identities, $x_1, x_2 \in \mathcal{V}^{int}$ and $\{x_1 + 1, x_2\} \in 2^{\mathcal{T}}$. The set of terms is mapped as a node to the same value. This mapping is explained in the next definition. The consequence is that this node implicitly expresses the relation $x_1 + 1 = x_2$.

Definition 2.4.4. Morphisms
A graph g has a *morphism* to a graph g' if there is a structure-preserving mapping from the nodes and the edges of g to the nodes and the edges of g' respectively. A graph g has a partial morphism to a graph g' if there are elements in g without an image in g'.

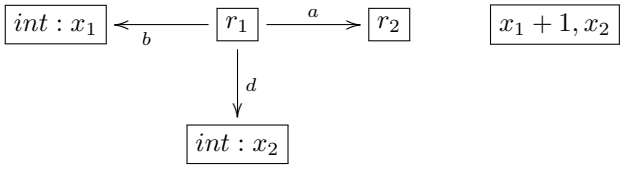


Figure 2.7: An example of a rule graph

Definition 2.4.5. Images
Given a morphism from a graph g to a graph g', a node or edge z in g has an *image* z' in g'. z is then a *pre-image* of z'. For each type of node, an explanation is given:

- A variable $v \in \mathcal{V}^s, s \in S$, has an image i in a host graph if $i \in \mathbb{U}^s$.
- A node $z \in 2^{\mathcal{T}}$ has an image i in a host graph if i is the valuation of all terms in z.

Definition 2.4.6. Transformation rules
A transformation rule r is a tuple $\langle LHS, NAC, RHS, l \rangle$, where:

- *LHS* is a rule graph representing the left-hand side of the rule
- *NAC* is a set of rule graphs representing the negative application conditions
- *RHS* is a rule graph representing the right-hand side of the rule
- $l \in L$ is the label of the rule

There exist implicit partial morphisms from the *LHS* to each rule graph in *NAC* and from the *LHS* to the *RHS* by means of the node identities. These morphisms are *rule graph morphisms.*

Definition 2.4.7. Creator and eraser edges
A *creator* edge is an edge in the *RHS* of a rule, that is not in the *LHS* of the rule. An *eraser* edge is an edge in the *LHS* of a rule that is not in the *RHS* or a rule.

Definition 2.4.8. Rule matches
A rule r has a *rule match* on a host graph G if its *LHS* has a morphism in G and none of the graphs in *NAC* have a morphism in G. Formally, $\nexists n \in NAC.nhasmorphisminG$. The morphism of the *LHS* to a host graph is a *match morphism.*

Definition 2.4.9. Rule transitions
After the rule match is found, all nodes and edges in *LHS* that do not have an image in *RHS*, are removed from G. All elements in *RHS* that do not have a pre-image in *LHS*, are added to G. The result of this process is called a *rule transition*, denoted by: $G \xrightarrow{r,m} G'$, where $m \in M$ is the morphism of the *LHS* to G.

Figure 2.8 shows an example of the initial graph G_0, one rule of a GG and the corresponding rule match. G_0 can be represented by $\langle \{n1, n2\}, \{\langle n1, a, n1\rangle, \langle n1, A, n2\rangle, \langle n2, B, n2\rangle\}\rangle$. The *LHS* of the rule has a match in G_0. Neither *NAC1* and *NAC2* have a match in G_0, because the edge with label C does not exist in G_0. The new graph after applying the rule is G_1.

Definition 2.4.10. Graph Grammars
A graph grammar is a tuple $\langle R, G_0 \rangle$, where:

- R is a set of graph transformation rules
- G_0 is the initial graph

By repeatedly applying graph transformation rules to the start graph and all its consecutive graphs, a GG can be explored to reveal a *Graph Transition System* (GTS). This transition system consists of graphs connected by rule transitions.

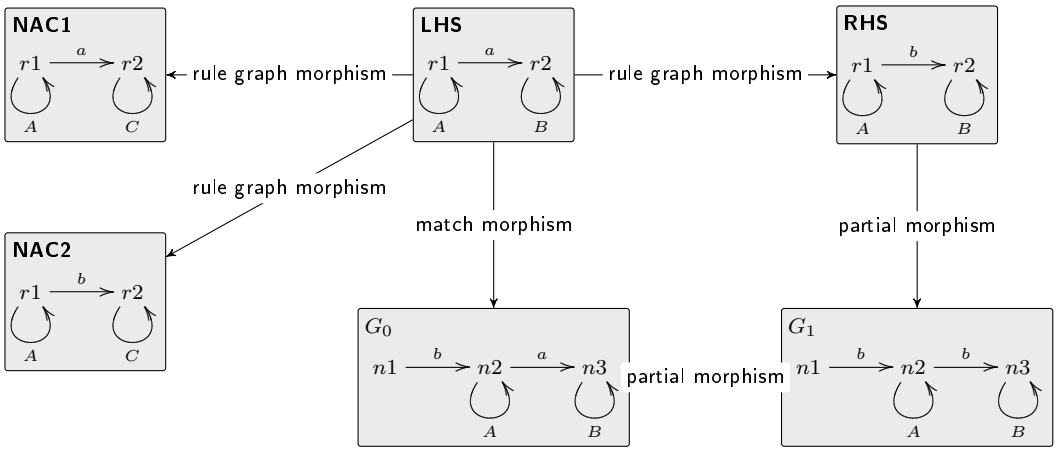


Figure 2.8: An example of a graph transformation

Definition 2.4.11. Graph Transition Systems
A graph transition system is a tuple $\langle \mathcal{G}, R, M, U, G_0 \rangle$, where:

- $\mathcal{G}$ is a set of graphs
- $L \in R \times M$ is a set of labels
- $U \in \mathcal{G} \times L \times \mathcal{G}$ is the rule transition relation
- $G_0 \in \mathcal{G}$ is the initial graph

Let $K = \langle R, G_0 \rangle$. A GTS $O = \langle \mathcal{G}, R, M, U, G_0 \rangle$ is derived from K by the following. $\mathcal{G}, M, U$ are the smallest sets, such that:

- $G_0 \in \mathcal{G}$
- if $G \in \mathcal{G}$ and $G \xrightarrow{r,m} G'$ then $G' \in \mathcal{G}, (r, m) \in L, (G \xrightarrow{r,m} G') \in U$

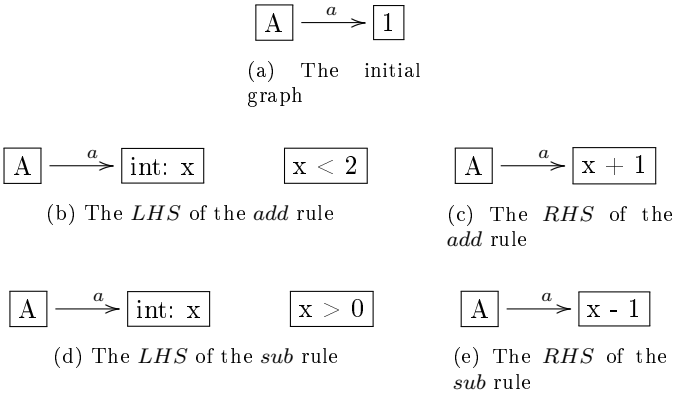
Definition 2.4.12. Input-Output Graph Grammars
In order to specify stimuli and responses with GGs, a definition is given for an *Input-Output GG* (IOGG). Concretely, the IOGG places input and output labels on its rule transitions. Following the definition from IOLTSs, each rule label $l \in L$ has a type $\iota \in Y$. Exploring an IOGG leads to an *Input-Output Graph Transition System* (IOGTS).

Definition 2.4.13. Rule priorities
A graph grammar with *rule priorities* P assigns a priority $p \in \mathbb{N}$ to a $r \in R$, such that $r \mapsto p \in P$. The definition of GTSs is extended with this definition of rule priorities and the following:
$r_1, r_2 \in R, G \in \mathcal{G}.(G \xrightarrow{r_1,m_1} G') \in U \wedge P(r_1) > P(r_2) \rightarrow (G \xrightarrow{r_2,m_2} G') \notin U$
Consider the graph grammar in Figure 2.9. The 'add' rule produces a rule transition to a graph, where the *sub* rule produces a rule transition back to the start graph. Suppose $P(add) > P(sub)$, then the 'sub' rule does not have a outgoing rule transition from the start graph.

2.5 Tooling

2.5.1 ATM

ATM is a model-based testing web application, developed in the Ruby on Rails framework. It has been used to test the software of several big companies in the Netherlands since 2006. It is under



(a) The initial graph

(b) The *LHS* of the *add* rule

(c) The *RHS* of the *add* rule

(d) The *LHS* of the *sub* rule

(e) The *RHS* of the *sub* rule

Figure 2.9: Priority rules

continuous development by Axini.

The UML sequence diagram for ATM is shown in Figure 2.10.

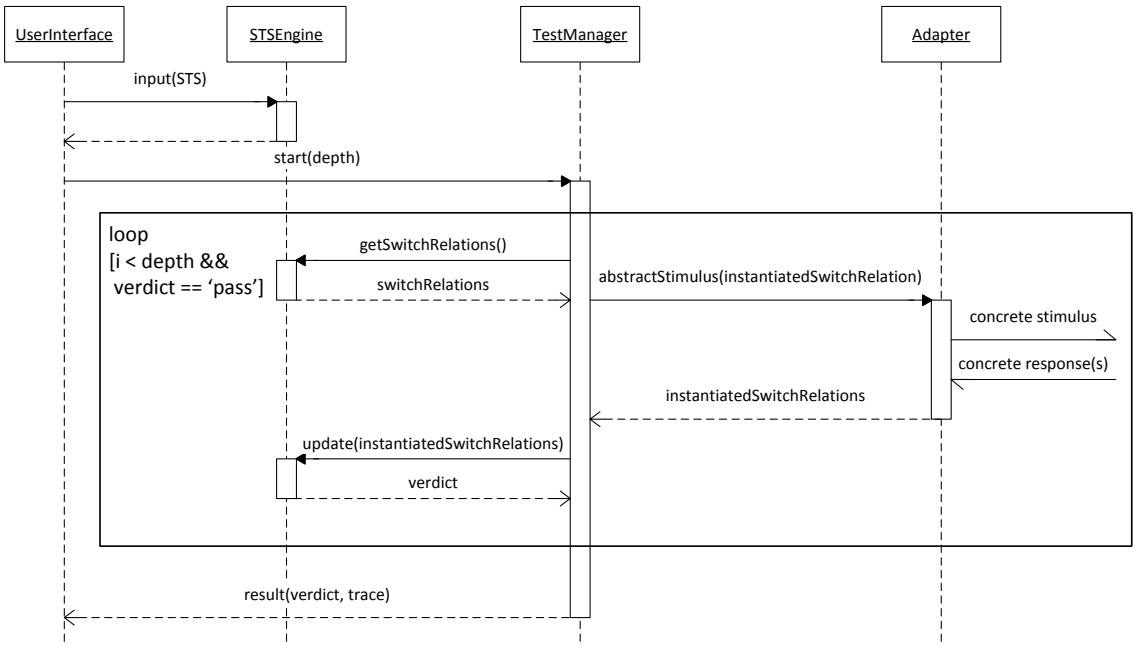


Figure 2.10: ATM sequence diagram

The tool functions as follows:

1. An STS is given to an STS Engine, which keeps track of the current location and variables. The user starts the test and gives a 'depth', indicating how many stimuli should be tested. The variable i stands for the current iteration of *loop* and is initially set to 0. The variable *verdict* is initially set to $'pass'$.

2. The STS Engine gives the possible switch relations from the current location to the Test Manager. It chooses an enabled switch relation based on a test strategy, which can be a random strategy or a strategy designed to obtain a high location/switch relation coverage. The valuation of the variables in the guard are also chosen by a test strategy, which can be a random strategy or a strategy using boundary-value analysis. The choice is represented by an instantiated switch relation.

3. The instantiated switch relation is given to the Test Execution component as an *abstract stimulus*. The term abstract indicates that the instantiated switch relation is an abstract representation of some computation steps taken in the SUT. For instance, a transition with label '?connect' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT.

4. The translation of an abstract stimulus to a concrete stimulus is done by the Adapter. This component has to be programmed by the tester such that the abstract stimulus is correctly translated to a concrete stimulus. This component provides the stimulus to the SUT. When the SUT responds, the Adapter translates this response to an abstract response. For instance, the Adapter receives an HTTP response that the TCP connect was succesful. This is a concrete response, which the Adapter translates to an abstract response, such as an instantiated switch relation with gate '!ok'. The SUT can also give multiple responses. As with the stimuli, the tester has to program the translation from concrete responses to abstract responses. The Test Manager is notified with these abstract responses.

5. The Test Manager updates the STS engine with the chosen abstract stimuli and received abstract responses. If this is possible according to the STS, a pass verdict is given, otherwise a fail verdict is given. The Test Manager updates the *verdict* variable accordingly. The loop continues as long as all responses are according to the specification and the required number of tested stimuli has not been reached. The test is stopped at a fail verdict, because the SUT has entered an invalid state and the STS engine cannot give possible switch relations any more. For instance, an error could have occurred, which is an invalid response and makes continuing impossible.

6. When the Test Manager finishes it gives a pass verdict for this test if all verdicts given by the STS engine were a pass verdict. Otherwise, the result is a fail verdict. Also a trace is given of all chosen and observed instantiated switch relations. This can be used to calculate coverage information for the test and to allow the SUT or the STS to be fixed in case of a fail verdict.

2.5.2 GROOVE

GROOVE is an open source, graph-based modelling tool in development at the University of Twente since 2004 [22]. It has been applied to several case studies, such as model transformations and security and leader election protocols [6].

The UML sequence diagram for GROOVE is shown in Figure 2.11.

The tool functions as follows:

1. A graph grammar is given as input to a RuleApplier component, which determines the possible rule transitions.

2. The user starts an ExplorationStrategy. This strategy explores all possible graph states and rule transitions. The possible rule transitions from the initial graph state are obtained from the RuleApplier and a rule transition is chosen, based on the exploration strategy. The target graph state of the chosen rule transition is again given to the RuleApplier until no more new graph states can be explored.

3. The ExplorationStrategy returns the explored GTS to the UserInterface.

2.5.3 GROOVE visual elements

Labels and flags Nodes in GROOVE have several kinds of labels: regular labels, type labels and flags. Figure 2.12 shows a node with a type label (bold), two flags (italic) and two regular

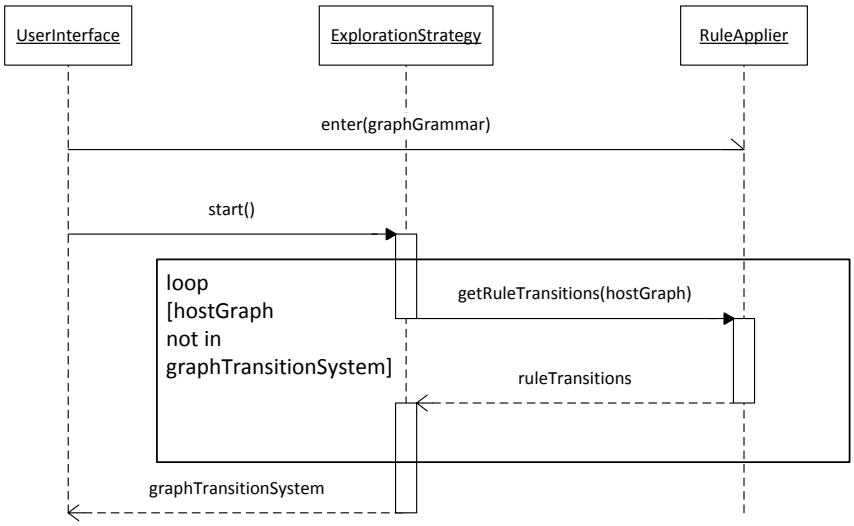
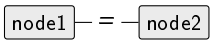


Figure 2.11: GROOVE sequence diagram

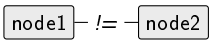
labels. Nodes in GROOVE can have one type, indicated by the type label. Typing is not explained further in this report[1]. A node can have multiple regular labels and flags.

Figure 2.12: GROOVE labels and flags

Rule node matching Nodes in a rule graph in GROOVE can also match each other, by connecting them with an equals '=' labelled edge. This means that any images of both nodes have to be the same. Figure 2.13a shows an example of this. Nodes in a rule graph in GROOVE can also explicitly not match each other, by connecting them with an not-equals '!=' labelled edge. This means that any images of both nodes cannot be the same. Figure 2.13b shows an example of this.



(a) Matching rule nodes



(b) Non-matching rule nodes

Figure 2.13: Node matching in GROOVE rule graphs

Colors Rule graphs in GROOVE combine *LHS*, *RHS* and *NAC* into one rule graph. The colors on the nodes and edges in GROOVE rules represent whether they belong to the *LHS*, *RHS* or *NAC* of the rule. See Figure 2.14 for an example.

1. normal line (black): This node or edge is part of both the *LHS* and *RHS*.
2. dotted line (red): This node or edge is part of the *NAC* only.
3. thick line (green): This node or edge is part of the *RHS* only.

dotted line is niet helemaal correct beter woord?

[1]See the documentation of GROOVE for more information: http://groove.cs.utwente.nl/doc/

4. dashed line (blue): This node or edge is part of the *LHS* only.

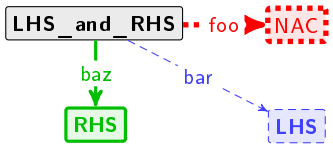


Figure 2.14: GROOVE rule graph colors

Variable nodes and terms Variable nodes in GROOVE are represented by their type: 'int', 'bool', 'real' and 'string' for integer, boolean, real and string variables respectively. Figure 2.15 shows two integer variable nodes and the constant integer node '1'. The diamond shaped node is a term node. It has two *argument* edges π_0, π_1 and a *result* edge 'int:add'. The term represented here is the addition of two integers, the first one being an integer variable, the second being the number 1. When this rule matches a host graph, the target variable node of the result edge is set to the result of the term; in this case the image of the first variable node plus one.

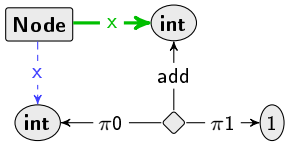
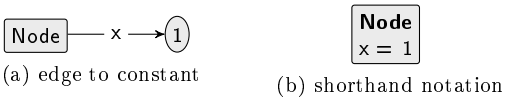


Figure 2.15: Terms in GROOVE rule graphs

Term shorthand notation A rule node with an edge to a constant is shortened to a label on the node. Figure 2.16a shows a node with an edge labelled 'x' to the constant '1'. Figure 2.16b shows the shorthand notation of this edge as the label 'x = 1' on the source node of the edge. Terms can also be shortened. The rule graph in Figure 2.15 can be shortened to the rule graph in Figure 2.17.



(a) edge to constant (b) shorthand notation

Figure 2.16: Constant shorthand notation in GROOVE

Rule transition parameters Rule transitions can have parameters in GROOVE. Figure 2.18a shows a rule where the node variables have a number in the top right of the node. These numbers indicate that the value of the variables are placed as parameters on the rule transition, in the order indicated by the numbers. This rule matches the host graph in Figure 2.18b. The result of applying the rule twice is the GTS in Figure 2.18c.

Quantification GROOVE supports quantification operations over nodes in rule graphs. Figure 2.19 shows a simple example. The 'forall' operator here matches all nodes typed 'Node'. GROOVE also supports the 'exists' operator and nesting of operators, however this is out of scope for this report. The 'forall' operator will be used in the model examples to perform operations over sets of nodes, such as in this rule: all self-edges labelled 'x' on nodes typed 'Node' are deleted from the host graph.

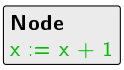
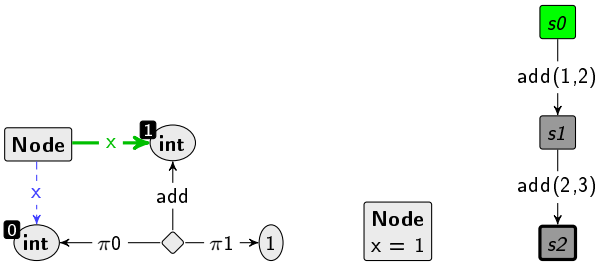


Figure 2.17: Term shorthand notation in GROOVE



(a) Rule graph (b) Host graph (c) GTS

Figure 2.18: Rule transition parameters in GROOVE

2.5.4 Example GROOVE graph grammar

The running example from Figure 2.4 is displayed as a graph grammar, as visualized in GROOVE, in Figure 2.20. The *LHS*, *RHS* and *NAC* of a rule in GROOVE are visualized together in one graph. Figures 2.20b, 2.20c and 2.20d show three rules. Figure 2.20a shows the start graph of the system.

The rules can be described as follows:

1. 2.20b: 'if a player has the turn and he has not thrown the die yet, he may do so.'
2. 2.20c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'
3. 2.20d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The graph is transformed after the rule is applied. The resulting graph after the transformation is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state. Figure 2.21 shows the IOGTS of one *?throws* rule application on the start graph. Note that the *?throws* is an input, as indicated by the '?'. State s_1 is a representation of the graph in Figure 2.20a. Figure 2.22 shows the graph represented by s_2.

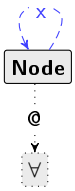
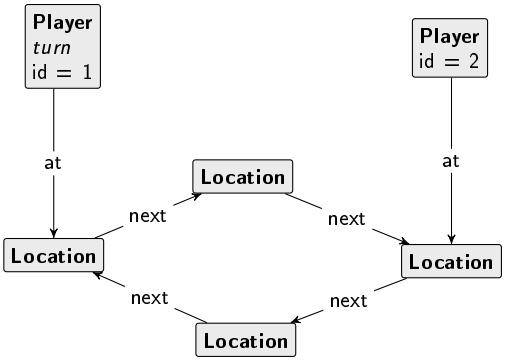
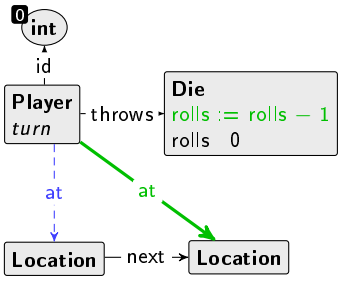


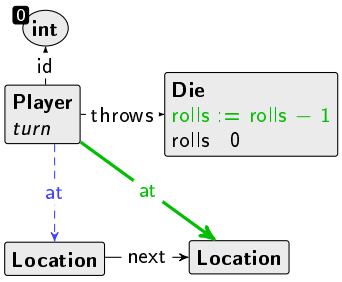
Figure 2.19: An example of quantification in GROOVE



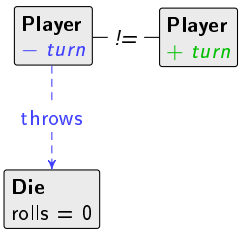
(a) The start graph



(b) The throw rule



(c) The move rule



(d) The next turn rule

Figure 2.20: The graph grammar of the board game example in Figure 2.4

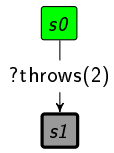


Figure 2.21: The GTS after one rule application on the board game example in Figure 2.20

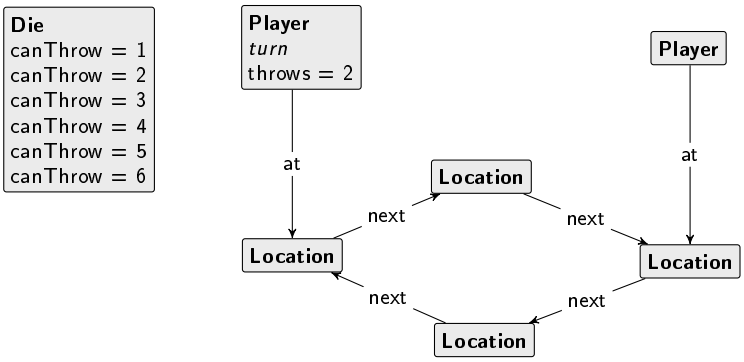


Figure 2.22: The graph of state $s2$ in Figure 2.21

Bibliography

[1] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1 – 54, 1999.

[2] Axel Belinfante. JTorx: A tool for on-line model-driven test derivation and execution. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 266–270. Springer Berlin / Heidelberg, 2010.

[3] E. Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing, and Verification VIII*, 1988.

[4] Duncan Clarke, Thierry Jéron, Vlad Rusu, and Elena Zinovieva. STG: A symbolic test generation tool. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *Lecture Notes in Computer Science*, pages 151–173. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-46002-0_34.

[5] D.M. Cohen, S.R. Dalal, M.L. Fredman, and G.C. Patton. The aetg system: an approach to testing based on combinatorial design. *Software Engineering, IEEE Transactions on*, 23(7):437 –444, jul 1997.

[6] A. H. Ghamarian, M. J. de Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *International journal on software tools for technology transfer*, 14(1):15–40, February 2012.

[7] Hasan and Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311 – 325, 1992.

[8] R. Heckel. Graph transformation in a nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1 SPEC. ISS.):187–198, 2006.

[9] M.R.A. Huth and M. Ryan. *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2000.

[10] C. Laird, L. & Brennan. *Software measurement and estimation: A practical approach*. IEEE Computer Society/Wiley, 2006.

[11] Tim A.C. Willemse Lars Frantzen, Jan Tretmans. Test generation based of symbolic specifcations. Technical report, Nijmegen Institute for Computing and Information Sciences (NIII), 2005.

[12] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996.

[13] R. Busser M. Blackburn and A. Nauman. Why model-based test automation is different and what you should know to get started. *International Conference on Practical Software Quality and Testing*, 2004.

[14] Joost-Pieter Katoen Manfred Broy, Bengt Jonsson and Alexander Pretschner. *Model-Based Testing of Reactive Systems.* Springer-Verlag Berlin Heidelberg, 2005.

[15] Thomas J. McCabe. Structured testing: A software testing methodology using the cyclomatic complexity metric. *National Bureau of Standards, Special Publication*, 1982.

[16] Steve McConnell. Software quality at top speed. *Softw. Dev.*, 4:38–42, August 1996.

[17] Lev Nachmanson, Margus Veanes, Wolfram Schulte, Nikolai Tillmann, and Wolfgang Grieskamp. Optimal strategies for testing nondeterministic systems. *SIGSOFT Softw. Eng. Notes*, 29:55–64, July 2004.

[18] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[19] Martin Pol. *Testen volgens Tmap (in dutch, Testing according to Tmap).* Uitgeverij Tutein Nolthenius, 1995.

[20] S.J. Prowell. JUMBL: a tool for model-based statistical testing. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, page 9 pp., jan. 2003.

[21] A. Rensink. Towards model checking graph grammars. In S. Gruner and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS), Southampton, UK*, volume DSSE-TR-2003-02 of *Technical Report*, pages 150–160, Southampton, 2003. University of Southampton.

[22] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer Verlag.

[23] Vlad Rusu, Lydie du Bousquet, and Thierry Jéron. An approach to symbolic test generation. In Wolfgang Grieskamp, Thomas Santen, and Bill Stoddart, editors, *Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, pages 338–357. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-40911-4_20.

[24] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003.

[25] Jan Tretmans. *A Formal Approach to Conformance Testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[26] Jan Tretmans. Test generation with inputs, outputs, and quiescence. In Tiziana Margaria and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 127–146. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-1_42.

[27] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2011.

[28] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39–76. Springer Berlin / Heidelberg, 2008.

[29] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.

List of Symbols